Error-bounded Lossy Compression of Floating-Point Color Buffers using Quadtree Decomposition

Jim Rasmusson · Jacob Ström · Tomas Akenine-Möller

Abstract In this paper, we present a new color buffer compression algorithm for floating-point buffers. It can operate in either an approximate (lossy) mode or in an exact (lossless) mode. The approximate mode is error-bounded and the amount of introduced accumulated error is controlled via a few parameters. The core of the algorithm lies in an efficient representation and color space transform, followed by a hierarchical quadtree decomposition, and then hierarchical prediction & Golomb-Rice encoding. We believe this is the first lossy compression algorithm for floating-point buffers, and our results indicate significantly reduced color buffer bandwidths and negligible visible artifacts.

Keywords Color Buffer Compression · Lossy Compression · High Dynamic Range · Real-Time · Texture Compression · Quadtree

1 Introduction

Rendering is essentially an *approximative* process, where an algorithm attempts to compute an image, which may resemble a photograph, for example. The algorithm may use different techniques that simulate how photons interact with an environment, and this simulation already includes a certain measure of approximation. In addition, since computers are used, there will be small approximation errors due to that floating-point numbers are used in the calculations, and at the end of the rendering pipeline, these floating-point values are often quantized to eight bits per component. Over the years, a number of *lossy* (i.e., approximative) texture compression techniques have been developed and broadly adopted in the real-time graphics industry. In addition, all precomputed radiance transfer algorithms [7] use some basis, e.g., spherical harmonics or wavelets, to approximately represent functions in. Furthermore, Monte Carlo rendering techniques use sampling to approximatively evaluate some kind of integral. These examples make it clear that approximative algorithms can be and are being used successfully in rendering.

Lossy techniques have also been used for color buffer compression and decompression [13]. However, that work targeted only low dynamic range (LDR) buffers, where each color component only has eight bits. More and more rendering is done directly to floating-point buffers, which can contain high dynamic range (HDR) content, and still, we have not seen any color buffer compression/decompression algorithms that are *lossy* (approximative) for such data.

Therefore, we present a new algorithm for lossy compression and decompression for floating-point color buffers. Our goals include a system where the introduced errors are kept under strict control, with substantially lower memory bandwidth usage, high quality of the rendered images, and reasonably low implementation complexity. This new algorithm builds on the work presented by Ström et al. [19] and by Rasmusson et al. [13] but contains several new additions each which significantly

J. Rasmusson
Lund University, Department of Computer Science
and Ericsson AB
Lund, Sweden
E-mail: jim.rasmusson@cs.lth.se
· J. Ström
Ericsson AB
Stockholm, Sweden
E-mail: jacob.strom@ericsson.com
· T. Akenine-Möller
Lund University, Department of Computer Science
Lund, Sweden
E-mail: tam@cs.lth.se

contributes to the quality and performance of the total buffer compression/decompression system:

1. The hierarchical quadtree decomposition has been extended with an adaptive flatness test during the hierarchical subdivision, which provides consistent introduced error per pixel. In the same spirit, we have developed a quantizer which adapts to the hierarchy level. These two techniques significantly reduce block artifacts.
2. The combination of using a decorrelating and reversible color transform together with an integer representation gives us the possibility to compress both lossily and losslessly in the same framework, which is important since it reduces the complexity of the system (if one want to support both lossy and lossless compression).
3. Hierarchical prediction.
4. Adaptive error thresholding: Detection of when high enough accumulated error has been reached, and in this case, subsequent compression is made to compress less aggressively, which improves quality and performance.
5. Constant bit rate mode, which can be used for HDR texture compression.

Next, we review previous work.

2 Previous Work

In this section, we present the most relevant work and their relations to our new algorithms.

Texture compression (TC) is a technique that is heavily used in real-time graphics today. This line of research started in 1996 [1,6,22], and since then, two major algorithms have been adopted for implementation in graphics hardware and supported by the APIs. S3TC (called DXTC in DirectX) is a collection of formats targeting both OpenGL and DirectX, while ETC [17] is used in OpenGL ES. The majority of these schemes compress to a fixed rate per block of pixels in order to simplify random access. Both S3TC and ETC use 4×4 pixel blocks and compress down to four bits per pixel (bpp). These algorithms are therefore, by design, *lossy*. It should be noted that most algorithms for TC are highly asymmetric, which in this case means that compression often can be done offline once, and it is only the decompression process, which is supported in hardware, that needs to be fast and of low complexity. The techniques above operate on low dynamic range (LDR) data only.

To support high dynamic range (HDR) textures, where each color component may be represented using a floating-point number (e.g., using 16 bits), specialized algorithms for HDR texture compression were introduced by Munkberg et al. [11] and Roimela et al. [16]. These also operate on 4×4 pixel blocks, and compress down to eight bpp. Sun et al. [20] and Wang et al. [23] take a different approach and implement an HDR TC algorithm using existing hardware for S3TC. This is in contrast to the other HDR TC algorithms which need new hardware for efficient decompression. There have also been some recent developments to the first algorithms, and both of these increase the image quality [12,15].

Textures are mostly read-only, and therefore it works well with asymmetric compression/decompression algorithms where compression is slower than decompression. However, the many buffers (e.g., color, depth, stencil, etc) in a real-time rendering pipeline must be treated differently for a variety of reasons. First, during the rendering of a triangle, compression and decompression may be applied several times, and hence these algorithms must be more symmetric. This also means that both compression and decompression must be supported by the hardware. See the surveys on both color buffer compression [13] and depth buffer compression [5]. In many cases, the user may need a buffer result which is lossless, i.e., exact. However, as argued in the introduction, approximate (lossy) algorithms is an interesting option. The major reason is that they offer considerably higher compression ratios, and we note again that lossy techniques are already in use in several different places in the pipeline. Rasmusson et al. [13] propose to use a lossy buffer compression / decompression technique where the introduced error is kept under control. When the error grows too large, the method reverts to using lossless (exact) compression or no compression. That algorithm operated only on LDR data. Recently, a lossless algorithm for color and depth buffer compression of floating-point data has been proposed by Ström et al. [19]. Similarly to the other buffer compression algorithms, this technique is also block-based to provide random access, and there is a fall-back to using no compression in order to be able to guarantee an exact result at all times. However, that work does not investigate approaches for *lossy* compression/decompression of *floating-point buffers*, which is the topic of the research presented in this paper.

Note that in principle, one can use existing lossy floating-point image compression algorithms, such as the B44A codec included in OpenEXR [2], for lossy buffer compression. The B44A codec is based on delta modulation and works in a way similar to the HDR-texture compression method by Roimela et al. [16], and we have included it in our results section in order to benchmark our algorithm.

3 New Color Buffer Compression Algorithm

In this section, we present a new color buffer compression method. The algorithm operates on tiles, which are typically 8×8 pixels. It is designed to compress 16-bit floating-point (fp16) high dynamic range (HDR) color buffers, although adapting the algorithms to 32-bit floaint-point numbers is straightforward. All operations related to the compression algorithm are done in the integer domain which lowers the complexity down to a level where it is within reach also for a mobile phone implementation. Note that while the bandwidth is significantly reduced due to the compression, the amount of storage in external RAM is not affected. This is due to that the algorithm uses variable bit rate *and* have an uncompressed mode as fallback, which is used for tiles that cannot be compressed using implemented algorithms (or if the maximum allowed amount of error is reached for this tile). See the papers by Hasselgren & Akenine-Möller [5] and Rasmusson et al. [13] for detailed descriptions on how buffer compression/decompression systems work.

The same implementation can be configured to operate in 8-bit integer low dynamic range (LDR) mode or in 16-bit floating-point (fp16) high dynamic range (HDR) mode. This is possible since we reinterpret the floating-point numbers as integers and do all operations related to the compression algorithm in the integer domain. It can be configured to operate in an approximate (lossy) or exact (lossless) mode with fine-grained control to go from one mode to the other. The approximate mode has mechanisms to automatically go from lossy to lossless mode when certain error thresholds have been reached. This effectively bounds the introduced errors to configurable maximum levels.

For each *tile*, a high level description of our algorithm is as follows:

1. Reinterpret the bit-pattern of the floating-point number of each color component as an integer, and transform (losslessly) from RGB into the decorrelated YC_oC_g color space.
2. Perform hierarchial quadtree decomposition.
3. Perform hierarchical prediction, quantization, and Golomb-Rice encoding (adaptive).

In addition, we have an error-bounded control mechanism to keep the introduced approximations under a user-defined threshold.

Our new method builds upon the color buffer compression methods by Ström et al. [19] and by Rasmusson et al. [13]. The color buffer is divided into 8×8 pixel tiles, and we also use a simplistic method to handle destination alpha, which is assumed to be 1.0 in the compression stage. If destination alpha $\neq 1.0$, or if any fp16 value is negative for any pixel in the tile, we simply revert to uncompressed mode. Alternative ways of handling alpha is to let the new compression algorithm presented in this paper also handle the alpha component, or let any existing scheme, such as DXTC or table-based methods [18], handle alpha compression for tiles with alphas between 0.0 and 1.0. At this point, we have omitted such studies since it would not advance the research field much. In addition, we leave the investigation of negative floats for future work, mostly due to the difficulty for us to find reasonable test scenes with negative values.

In the following, we describe the steps (1–3 above) in detail.

3.1 Representation and Color Space Transform

Without an excessive numbers of bits, it is impossible to represent the difference between two arbitrary floating-point numbers in a lossless manner. For positive floating-point numbers, we can circumvent this problem by taking the bit pattern of the floating-point number and interpreting them as an integer [8,19].

After R, G, and B for each pixel in a tile have been reinterpreted as integers, we convert these into a luminance/chrominance color space. Besides decorrelation of the RGB channels, this enables the possibility to have different compression settings for the luminance and the chrominance components, respectively. For example, in a lossy mode, we typically compress the chrominance components more aggressively than the luminance components. This will introduce higher error levels in the chrominance components, but since the human visual system is less susceptible to chrominance errors than luminance errors, the resulting visual artifacts are less visible. Introducing higher error levels in the chrominance than the luminance components are common in most image and video processing methods, such as those in JPEG and MPEG.

The color space transform we use is the YC_oC_g-transform [9], which has some very attractive properties. It has low implementation complexity (highly silicon-efficient), and provides good decorrelation which in general reduces bandwidth. Note that since we operate on pixel component values that are remapped from floating-point to integer, the behavior of this transform will not be the same as the original YC_oC_g-transform [9]. We have not done any extensive analysis of how the decorrelation properties change after the remapping operation. However, we have observed that it brings performance improvements on par with what it brings in the LDR domain (about 10% bandwidth reduction).

Another property important for us is that this transform is reversible; transforming the YC_oC_g-values back to RGB recreates the RGB values bit-exactly (that is, if the YC_oC_g-values have not been altered). This is a must for our lossless mode.

Transforming from RGB to YC_oC_g and back is done as follows. Note that the R, G, and B numbers are 15-bit integers (the sign bit is assumed to be zero), and all operations work on integers.

$$\begin{aligned} C_o &= R - B \\ t &= B + (C_o >> 1) \\ C_g &= G - t \\ Y &= t + (C_g >> 1). \end{aligned} \tag{1}$$

The reverse transforming is as simple:

$$\begin{aligned} t &= Y - (C_g >> 1) \\ G &= C_g + t \\ B &= t - (C_o >> 1) \\ R &= B + C_o. \end{aligned} \tag{2}$$

The Y-component has 15 bits, and the C_o and C_g-components have 16 bits respectively. The color space transform has been used for LDR color buffer compression before [13], and the reinterpretation of floats as integers has been used before as well [8, 19]. However, for us, it is crucial to use a luminance and chrominance divided color space since it allows us to compress the chrominances stronger than the luminance. The decorrelation also improves compression performance (in general around 10% bandwidth reduction). Furthermore we must be able to support both lossless and lossy compression of floating-point data. The use of this combination is a small, albeit very useful and practical contribution in this context.

3.2 Hierarchical Quadtree Decomposition

The next stage is a quadtree decomposition of the transformed tile data. This creates a hierarchical tree structure of the 8×8 pixels, where "flat" (homogeneous) regions are sub-sampled and thus represented in higher levels in the tree, and using fewer samples than one per pixel. As an example, only one sample per 4×4 sub-tile may be used if that sub-tile is flat enough. For an 8×8 pixel tile, there are at most four levels, as can be seen in Figure 1.

However, it is important to avoid sub-sampling when the sub-tile region is not flat, e.g., when there are edges, as this can cause visible artifacts. We use a *flatness test*, where we first calculate the average of the 2×2 pixels in the sub-tile. If the absolute difference between each

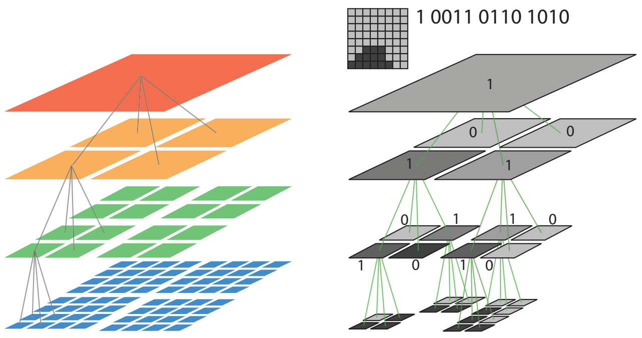


Fig. 1 Left: a full hierarchical quadtree decomposition of an 8×8 pixel tile. The bottom level has full pixel resolution (8×8 pixels), and each level above is a subsampled version of the level below. Right: in this example, we show the hierarchical decomposition of the 8×8 pixels (light and dark gray) at the top. Note that if a subtile has constant color, the representation ends at that level, and that is the reason why the tree is not full. Note also that the associated 13-bit tree code is shown at the top, where 1's means that the subtile has children, and 0's means that the node is a leaf. Thus, the first bit (1) indicates that the root node (top) has children, and the first two of these children do not have children of their own (00), while the remaining two has (11), and so on.

pixel value and the average is below a configurable "flatness" threshold, τ_{flat}, we assume it is reasonable to use subsampling. This is done for all pixels in the tile in a bottom-up fashion. We start with the individual pixels and attempt to sub-sample each 2×2 pixel sub-tile, and then move up in the hierarchy until all four levels have been tested. See Figure 1. If the entire 8×8 pixel tile consists of a really flat region, a single value is sufficient to represent the entire tile. This case actually occurs surprisingly often for the chrominance components, C_o and C_g. Computer generated content often decorrelate well by the YC_oC_g-transform, and this results in a Y-channel carrying most of the information, while the chrominances are more uniform (flat).

Quadtree decomposition is a well known image and graphics processing technique that yields an efficient and compact data structure. It has often been in use for image compression [21, 24]. However, to the best of our knowledge, we have not seen this being used in any buffer compression and decompression algorithms.

In Figure 2, the contribution of the quadtree stage is shown in a rate-distortion plot. As can be seen, the addition of the quadtree mechanism to the algorithm brings a significant improvement in terms of reducing bandwidth while keeping the distortion reasonably low. This clearly motivates inclusion of quadtree decomposition in our system.

An associated binary tree code describes the structure of the tree in a compact way. For an 8×8 tile, the tree code varies between 1 and 21 bits. A full tree (where all nodes have four children) uses 21 bits (1+4+16), and

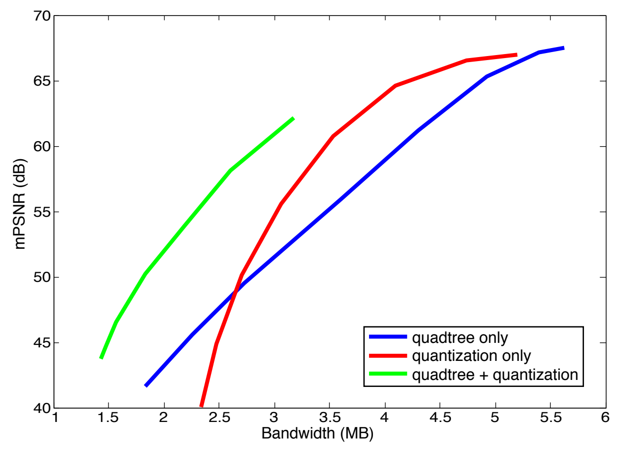
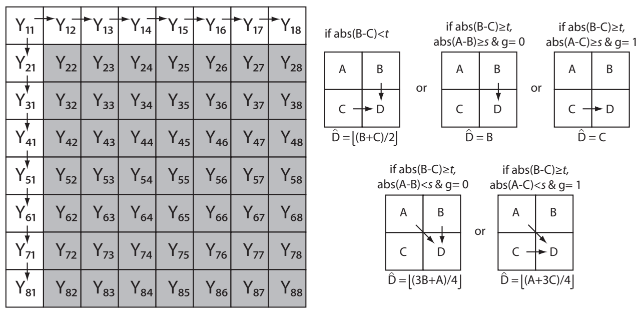


Fig. 2 Rate distortion plots of the two lossy stages in the algorithm; the quadtree stage and the quantization stage. The two rightmost curves are showing the two stages working in isolation, and the leftmost curve shows the combined effect. Note that the quadtree curve's slope is flatter while the quantization curve's slope is steeper when moving towards lower bandwidths. These results were made using a near exhaustive parameter search of the Shadows scene (see Table 1) at 1024 × 768 resolution. Note that our algorithm uses both the quadtree and quantization.

the smallest tree uses a single bit (i.e., indicating that the 8×8 pixel tile is sub-sampled to one value), as illustrated to the right in Figure 1. The tree code is stored in the compressed data.

Note that the flatness threshold, τ_{flat}, is lower the higher up in the hierarchy the current subtile is at. That is, in order to sub-sample a region covering more pixels, "the more flat" it has to be, if the introduced errors per pixel are to be consistent. This also reduces block artifacts. Since a pixel in level l covers four pixels in level $l-1$, we divide τ_{flat} by four (by right-shifting two steps) each time we move up to a higher level. Note also that the flatness threshold for the Y-components can be lower than the corresponding thresholds for the C_o and C_g-components. Again, the reason for this is that the human visual system is more susceptible to errors in the luminance components. Also, as mentioned before, most details are often present in the luminance component. For lossless compression, we simply set $\tau_{\text{flat}} = 0$, and the values in the sub-tile regions have to be exactly the same in order to be sub-sampled. This happens sufficiently often to justify the decomposition stage to be active also in our lossless mode. The flatness threshold is also dependent on the amount of accumulated error present from earlier approximations in the tile. See Section 4 for more information on the error control.



Y11 Y12 Y13 Y14 Y15 Y16 Y17 Y18
Y21 Y22 Y23 Y24 Y25 Y26 Y27 Y28
Y31 Y32 Y33 Y34 Y35 Y36 Y37 Y38
Y41 Y42 Y43 Y44 Y45 Y46 Y47 Y48
Y51 Y52 Y53 Y54 Y55 Y56 Y57 Y58
Y61 Y62 Y63 Y64 Y65 Y66 Y67 Y68
Y71 Y72 Y73 Y74 Y75 Y76 Y77 Y78
Y81 Y82 Y83 Y84 Y85 Y86 Y87 Y88
if abs(B-C)<t
if abs(B-C)≥t, abs(A-B)≥s & g= 0
if abs(B-C)≥t, abs(A-C)≥s & g= 1
A B C D
or
D̂ =⌊(B+C)/2⌋
D̂ = B
D̂ = C
if abs(B-C)≥t, abs(A-B)<s & g= 0
if abs(B-C)≥t, abs(A-C)<s & g= 1
D̂ =⌊(3B+A)/4⌋
D̂ =⌊(A+3C)/4⌋

Fig. 3 Arrows indicate prediction. For instance, Y_{12} is used to predict Y_{13}. Pixels marked in gray can be predicted from the pixels to the left, to the top, and/or top-left neighbors, as shown to the right. The correct D-value is predicted as $\hat{D}$ from pixels A, B, and C. See also Equation 4.

3.3 Prediction, Quantization, and Encoding

The next few steps are prediction, quantization, and finally encoding. These are presented together since they often are tightly connected in an implementation.

3.3.1 Prediction

For prediction, we use a modified version of the novel predictor used for color buffer compression [19]. A useful feature of this predictor is that it avoids prediction across edges. This is of particular importance during the rendering of computer generated content where sharp edged primitives (e.g. triangles) are drawn on top of other primitives or background images. Hence, the tiles that are to be compressed often contain high intensity transitions due to these edges. The predictor traverses the pixels in the tile from left to right, and in top to bottom order. See Figure 3. If the difference between pixels B and C is big enough, it is assumed that an edge has been found, and an extra guide bit, g, is used to indicate whether to predict from B or C. Otherwise, the average of B and C is used. For the gray pixels in Figure 3, the predictor [19] is summarized below:

$$\hat{D} = \begin{cases} \lfloor \frac{1}{2}(B+C) \rfloor, & \text{if } |B-C| < t \\ B, & \text{if } |B-C| \geq t, \text{and } g = 0 \\ C, & \text{if } |B-C| \geq t, \text{and } g = 1, \end{cases} \tag{3}$$

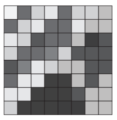
where $\lfloor \cdot \rfloor$ denotes rounding to the nearest lower integer, and t is a threshold value. In our experiments, we have found that the following modification improves performance slightly:

$$\hat{D} = \begin{cases} \lfloor \frac{1}{2}(B+C) \rfloor, & \text{if } |B-C| < t \\ B, & \text{if } |B-C| \geq t, |A-B| \geq s \,\&\, g=0 \\ C, & \text{if } |B-C| \geq t, |A-C| \geq s \,\&\, g=1 \\ \lfloor \frac{3}{4}B + \frac{1}{4}A \rfloor, & \text{if } |B-C| \geq t, |A-B| < s \,\&\, g=0 \\ \lfloor \frac{1}{4}A + \frac{3}{4}C \rfloor, & \text{if } |B-C| \geq t, |A-C| < s \,\&\, g=1. \end{cases} \tag{4}$$

In the new predictor in Equation 4, we added an extra test to check whether the information in pixel A is useful in the predicton. If A is similar enough to B and C, respectively, we include A when calculating $\hat{D}$. We have found that $s = 512$ works well in practice. Note that the multiplications by $1/4$ and $3/4$ are done using shifts and adds in order to reduce complexity. The threshold t is set to 2048 (same as Ström et al. [19]).

We traverse the pixels left-to-right and top-to-bottom. The first upper left corner pixel is always stored separately in uncompressed form.

To indicate where a new edge is first encountered, Ström et al. [19] use a restart value, which consists of a pixel position inside the tile, and the value at that pixel. The idea is that the first time a very different value occurs, it should be possible to state that explicitly. For the rest of the pixels, it should be possible to predict either from this new value or from the previous values. However, sometimes a block may contain more than two sets of very different values and a single restart value is not enough, as illustrated in the figure to the right.



To mitigate this situation, Ström et al. [19] have a rotation mechanism which changes the pixel traversal order, and checks if rotating the sub-tile yields a more favorable situation for the predictor. While the rotation helps, there are cases where it does not help and you end up predicting across edges, resulting in large "jumps" in the stream of residuals. This degrades the performance of the subsequent Golomb-Rice encoding.

We take a different approach to the use of restart values. While Ström et al. use a single restart value per tile and perform an exhaustive search [19] to find the best position for it in the tile, we propose to allow as many restart values as it takes to accurately represent an edge (or several edges) inside a tile. The restart values are also detected on-the-fly in the predictor and thereby we avoid the expensive exhaustive search method of Ström et al. [19]. Furthermore we also skip the rotation as it requires an extra encode session (to test if the rotated tile is better).

In order to find the restart values, we have added an extra test in the predictor that checks if the predicted value and the current value differs with more than τ_{restart} (another threshold value), i.e., $|D - \hat{D}| \geq \tau_{\text{restart}}$. In practice, we have found that $\tau_{\text{restart}} = 8192$ works well here. If this is the case, we generate a new restart value.

These restart values are basically random values, and so there is no point in using a predictor on these. Furthermore, the distribution of restart values typically does not suit the Golomb-Rice coder, and hence we code and store them separately. Pruning the restart values from the stream of residuals also has the added benefit that the Golomb-Rice coder performs well with a single divisor for all residuals, see Section 3.3.3.

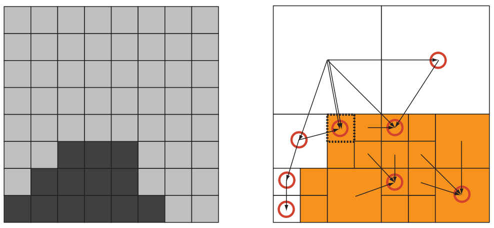


Fig. 4 Illustration of a few prediction cases using the example tile from Figure 1. Note that the sub sampled values are used for prediction. The number of predictions are defined by the quad tree. Arrows indicate prediction (only a few cases, marked with red circles are illustrated). Pixels marked in orange use the three neighbor pixels (left, top, and/or top-left) and white pixels use only one (either left or top). For some pixels, two out of the three neighbors are the same (e.g. the dotted pixel). See also Equation 4.

Note that we use the subsampled pixel values when calculating the prediction values. Due to the adaptive sub-sampling scheme during the quadtree creation, a varying number of pixels needs to be predicted. See Figure 4.

The predictor also needs to handle the odd cases where the top-left pixel, due to the sub-sampling scheme, is the same as the either the left or the top pixel. The dotted pixel in Figure 4 is an example of such a case. After subtracting the predicted values from the input values (subsampled), we have a number of residual values. A last important thing to realize is that the predictor on the encoding side must behave in the same way as on the decoding side. For this reason, the pixels are at all times modified to the decoded (lossy) values while predicting. This means that, in the predictor, we quantize the residual, store the value for further entropy encoding and then do an inverse quantization, to get the decoded value back.

The quantizer and the entropy coding scheme are described next.

3.3.2 Quantization

In the lossy mode, the residual values are quantized. We use a uniform quantizer where the quantization step is dependent on which sub-sampled area the current pixel is in. This means that the more flat the region, the more gentle quantization. This is done to reduce the amount of introduced errors with the motivation that these sub-sampled values represent more pixels, and therefore the amount of quantization should be reduced in order to be consistent. Next, the details of our quantization are described.

We first compute the quantization step as: $q_{\text{step}} = (q_{\text{level}} \gg l) \gg q_{\text{errorweight}}$, where $\gg$ denotes right shift, and q_{level} is the basic quantization level set by the user—the higher value, the more loss in the compression. To reduce block artifacts, the subsampling level, l, reduces the amount of quantization the higher up in the hierarchy the sub-tile is located, and the right shift by l should correspond to a division by the number of pixels covered by that subtile. For example, for the top level (which covers 8×8 pixels), the quantization level should be divided by 64, i.e., $l = 6$. More generally, assume there are $2^p \times 2^p$ pixels in the current subtile. The subsampling level is then computed as $l = 2p$. The $q_{\text{errorweight}}$ parameter is described in Section 4.1. Once q_{step} has been computed, the residual values, r, are quantized according to: $\hat{r} = r \gg q_{\text{step}}$. The quantized residual values, $\hat{r}$, are then subsequently entropy coded for further compression. In the following, we argue that it is reasonable to perform the quantization in the integer domain. Recall that the float-to-integer conversion is monotonic, and neighboring positive floats are converted to neighboring positive integers. Thus, converting a float to an integer, and quantizing the integer (ie., slightly perturbing the integer), is equivalent to a slight perturbation of the original floating-point value. Hence, it is clear that we can quantize in the integer domain, with expected results.

3.3.3 Encoding

We use the standard Golomb-Rice technique [4, 14] for entropy encoding. When the predictor generates residual values whose distribution has higher density around zero, good compression ratios can be expected since fewer bits are spent on smaller values in Golomb-Rice. A further advantage is that the implementation is of reasonably low complexity.

For clarity, we briefly recap how the Golomb-Rice code works, and refer to recent papers [13, 19] for a detailed explanation. Each residual value is divided by a divisor, 2^k, which creates a quotient and a remainder. The quotient is encoded using unary encoding and the remainder is coded using binary encoding. A zero is used to separate the two. We have found that a single divisor per tile yields the smallest bandwidth. This is possible due to that we store the first upper left corner value and the restart values separately, which makes the stream of residuals "pruned" and better suited for the Golomb-Rice coder.

To find the best k-value, we use a method [19], where the the bit position, p, of the most significant bit of the largest residual value is calculated. We then test the interval $[p-4, p]$ for the best k-value.

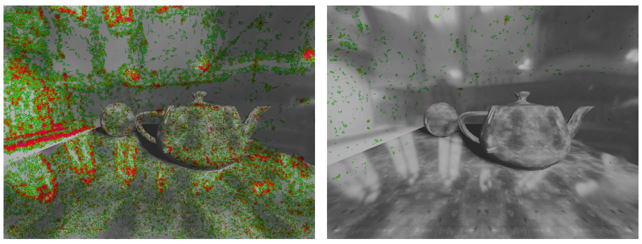


Fig. 5 In these images, we visaualize the error computed on the luminance channel using the HDR visual difference predictor (VDP). Green pixels are those where a human has a 75% chance of detecting the artifact, and red pixels have 95%. In the left image, *no* error control has been used, while in the right image the compression was error-bounded using our algorithm. All other parameters were the same. Clearly, there is a need for error control.

After this, the entire encoding is finished, and can be written over the bus to external memory.

4 Error Control

For *lossy* buffer compression, it is of utmost importance to have some sort of error control that prevents the accumulated error to grow out of reasonable proportions. The reason why this is needed is that compression can be applied many times to a tile (due to that many triangles can render to the same tile at different times), and if an error is introduced when the first triangle is rendered, then this error may grow when the second triangle is rendered, and the tile is compressed again. This is often called *tandem* compression. The difference between using and not using an error-bounded algorithm can be seen in Figure 5, where visible errors are visualized using the HDR visual difference predictor (VDP) [10].

We use an error control mechanism similar to the one proposed by Rasmusson et al. [13]. This algorithm gauges the introduced errors, and a decision is taken to use approximative compression if the newly introduced error and the previously accumulated error together are below a predetermined error threshold. If this condition is not met, then we revert to the lossless mode instead. The error metric we used is the mean square error (MSE) between the decoded RGB and the input RGB values. See Rasmusson et al's paper for more details.

4.1 Graceful Parameter Adaptation

We now present an improvement of the previous error control mechanism. A simple observation here is that the compression algorithm should revert to lossless encoding as seldom as possible. To that end, we introduce *graceful parameter adaptation*, so that the higher the accumulated error is in the tile, the lower is the flatness threshold and also the quantization step. This allows us to use lossy compression more often and it allows us to use more aggressive compression for tiles that are compressed only a few times.

The net effect is that the newly introduced approximation errors will be smaller and smaller the closer to the error threshold we get. More importantly, the algorithm can continue to introduce small errors, without having to revert to the much more expensive (in terms of number of bytes) lossless mode. In our current implementation, we detect when the accumulated error level has reached 50% of the error threshold, and in those cases, we divide τ_{flat} by a factor of two and change $q_{\text{errorweight}}$ (used in Section 3.3) from 0 to 1 . This could be extended so that when the accumulated error has reached 75% of the error threshold, the division factor is four instead of two and set $q_{\text{errorweight}} = 2$, and so on. A further extension would be to use the full "$1/x$"-behavior.

In general, this mechanism gives an improvement of $0.5 - 2$ dB in mPSNR, while the bandwidth usage remains constant or is even reduced by up to three percent. However, for certain tiles the bandwidth may go up. Intuitively, this may happen when the τ_{flat} is lowered due to that we reach the 50% threshold, and then nothing more is rendered to that tile, which means that there was no use in lowering the threshold. It should be noted that for those tiles, the image quality is increased though.

For our lossy mode, we also quantize the restart pixel values (not the position) to 8 bits each, but when we reach the 50% error level, quantization is instead changed to 12 bits.

5 Implementation

Our algorithm evaluation (see Section 6) was done in a software-based simulation framework implementing a tiled rasterizer with a modern color buffer architecture, programmable shading, texture caching, Z-culling, and more. We have used a a color buffer cache of 1 kB, and 256 bits for a tile table cache to store the tile table bits that indicate which compression mode is used (uncompressed, tile clear, compression mode 1 or compression mode 2).

In order for us to better understand the introduced error levels, we have used mean-square (MSE) errors between the incoming (original) tile and the same tile being encoded and decoded for the error control mechanism. This correlates well with established error metrics (e.g., mPSNR [11]) when we measure error on the resulting final image. However, for a hardware realization, methods of lower complexity could be developed. For example, error gauging can be made on-the-fly directly inside the quadtree decomposition stage and the predicton stage. Less expensive error metrics (in terms of complexity) such as sum-of-absolute-differences (SAD) may be used instead of MSE. This would need further analysis, and therefore, we leave this for future work.

While the properties of an algorithm, and its performance in terms of image quality and compression ratios can be evaluated using a software simulation, it is also important to investigate whether a hardware implementation is feasible. The different processing blocks have been designed with low complexity in mind. The YC_oC_g-transform consists solely of integer adders, subtracters and bit shifters and will incur a tiny cost in silicon. In order to reduce latency, many transform blocks can be executed in parallell. The quadtree decomposition method is also low complexity and highly parallelizable. In our predictor, a majority of the operations are bit shifters, adders and subtractors, and hence the complexity is relatively small.

We believe the total complexity of a combined compressor and decompressor will be low enough even for a silicon implementation in embedded battery-driven devices like mobile phones. This is supported in a Master thesis report by Caglar and Ojani [3] showing results from a silicon implementation of the lossless mode of the algorithm by Rasmusson et al. [13], including the rather complex variable bit rate parts. The resulting size of the compression and decompression blocks was well below 0.1 mm^2 in 65nm technology. That implementation operated on 8×8 pixel tiles, but only on 8-bit integer color (i.e., LDR), while we need to use 16-bit integers. Since mostly adders and shifts are used in our algorithm, we believe that an implementation in silicon will scale up in a decent way.So, while we have not implemented our algorithm in hardware exactly, the arguments above support that this will be feasible at a low cost considering that we are dealing with HDR colors.

6 Results

In Section 5, we describe the software simulation framework that we have used for algorithm evaluation. We emphasize the fact that the testing includes the full, incremental rasterization process of these scenes. This is in contrast to regular image compression, where only the final image is being compressed.

The test scenes are *Water*, *Shadows*, and *Reflections*, and all render targets are fp16 color buffers. To make certain that a large interval of the dynamic range is used, all scenes use fp16 texture maps and cube maps. In addition, the *Shadows* scene renders shadows using shadow mapping, and *Reflections* renders an fp16 cube map every frame. The sphere in the center of the scene is rendered using reflections with this dynamic cube map. Screenshots from these scenes can be seen in Table 1. Note that these screenshots have been tone mapped from fp16 RGBA down to 8 bit RGBA. This means that if a region in the screenshot appears to be simple to compress (e.g., a white or black area), this may not at all be so because the raw fp16 data can contain a lot of information even in those areas. Since we cannot know beforehand how the tone mapping operator works, we need to be able to compress even these areas with high quality.

In order to evaluate the error/quality of the final rendered images, we use HDR-VDP [10], logRGB, and mPSNR [11]. The HDR-VDP numbers presented in Table 1 are given after a manual adjustment using the multiply-lum command increasing the luminance level to 300 cd/m^2.

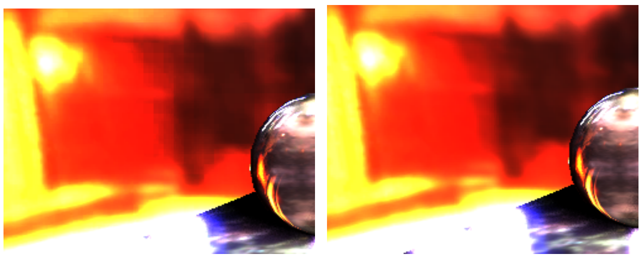


Fig. 6 Crops from the shadow scene at 1024×768. Left: block artifact reduction methods disabled. Right: block artifact reduction methods enabled. In this case, mPSNR is about 4 dB higher with block artifact reduction enabled when the parameters are tuned for equal memory bandwidth usage. To clearly see the block artifacts in the left image, it may be necessary to zoom in the pdf.

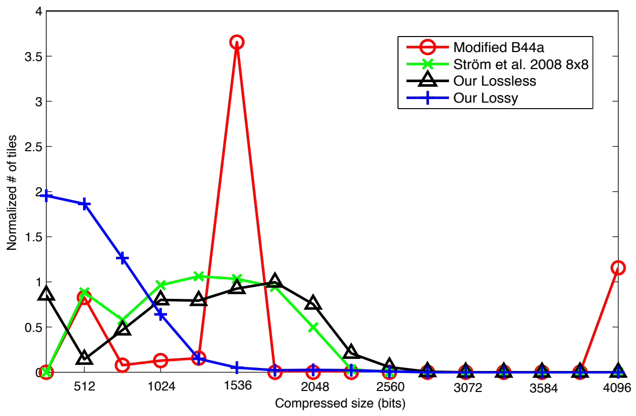


Fig. 7 A normalized histogram of the number of tiles that are compressed to a given size (256 bits bins), using different algorithms. We use 8 × 8 pixel tiles, which means that 4096 bits indicate uncompressed tiles. The histogram is based on an average over all our test scenes.

6.1 Contender Codecs for Benchmarking

We have compared our lossy compression algorithm against OpenEXR's B44A mode (described in Appendix A), both in terms of compression performance and quality. B44A is a lossy compressor, and it is the one that we found most amenable for hardware implementation. We have modified the B44A algorithm to include our error control mechanism. In addition, as a benchmark for our lossless method, we have used a modified version of the method from Ström et al. [19]. The original method operates on 8 × 8-blocks, but divides this into four 4 × 4 blocks in order to enable parallel encoding. This has a negative impact on the compression efficiency, and in order to provide a fair comparison, we have implemented an 8 × 8 version. It works in the same way, but only one base value and one restart value per 8×8 block are now used. Note that the image error/quality measures are only relevant for the B44A lossy compressor and our lossy compressor.

6.2 Block Artifact Reduction

Since we are working with lossy compression of 8 × 8 pixel tiles, there is a risk for block artifacts. For this reason, we have designed block artifact reduction techniques in two places in the algorithm. The first place is in the quadtree decomposition stage where the flatness thresholds are decreased by a factor of four for each higher level up in the quadtree. In the same spirit, the quantization stage is designed to lower the quantization step and apply milder compression the more subsampled a residual value is. See Section 3 for more details. In Figure 6, the effect of the block artifact reduction methods are shown.

6.3 Target Memory System

Since we do not know the target memory system, it is hard to predict what burst sizes are better than others. The memory system of a graphics card for a PC is dramatically different compared to that of a mobile phone. For this reason, the bandwidths and associated compression ratios shown in Table 1 are given without burst size limitations. It should be noted that these are unrealistically high. In order to present feasible burst sizes in a more memory system agnostic way, we use a histogram to show how often a particular "bit size bin" is used (vs bin sizes). See Figure 7.

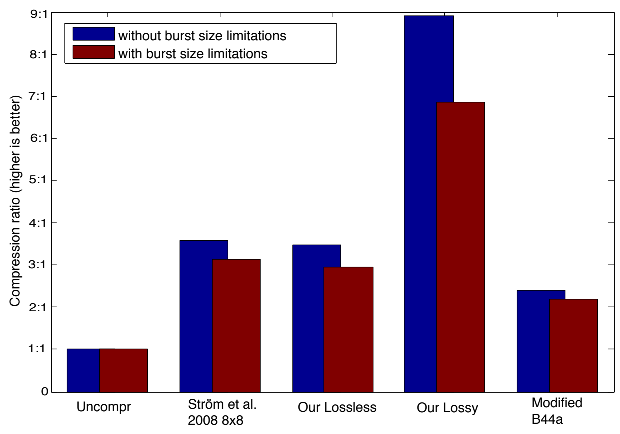


Fig. 8 Compression ratios with an optimal memory system (blue bars) and a more realistic example memory system (red bars) using a minimum burst size of 256 bits, with three tile bits (uncompressed, cleared, 256, 1024, 1536, 1792, 2048, 2560 bits). The diagram is based on an average over all our test scenes.

In our tile table, we may use two bits to indicate compression mode (uncompressed, tile clear, and two different compression modes) or we may use three tilebits (uncompressed, clear and six compression modes). With detailed histogram data, it is possible to analyze what compression modes would be appropriate for a particular target memory system. If you have, for example, a mobile phone system, with a 32-bit data bus and mobile DDRAM, burst sizes of n times 256 bits (8×32 bit words) typically make sense. With three tile bits, you have 6 compression modes to choose from. Analyzing the histogram data will give the six best burst sizes. The resulting compression ratios for this example is shown in Figure 8 (shown together with the compression ratios without burst size limitations).

6.4 Performance and Image Quality Evaluation

Table 1 shows the color bandwidth figures and the image quality/error measures for the the three scenes rendered at 320×240 and 1024×768 resolutions. Note that we have tuned the thresholds of our algorithm so that the image quality/error measures are about the same for both the B44A (a lossy compressor) and our lossy compressor. As can be seen, our algorithm always outperforms the B44A algorithm. Our research in this paper has not focused on lossless compression, but our compressor can be configured as a lossless compressor by setting $\tau_{\text{flat}} = 0$, and the error threshold to zero as well. This was done for the columns **C** in Table 1, and as can be seen, our results are about the same as the lossless compressor presented by Ström et al. [19]. It can be seen that the image error/quality measures of the rendered images have high quality overall, and hence, we believe that our algorithms make a significant contribution, since the compression factors are also rather high.

	Bandwidth (MB/frame)	mPSNR (dB)
H.264 16×16	0.397	49.1
Our 16×16	0.385	49.5
Our 8×8	0.295	49.5

Table 2 A comparison with the H.264 codec.

6.5 Comparison with H.264

In addition to comparing to B44A, we have also included a comparison against H.264. Although we guess that H.264 is too complex to be used for color buffer compression, we think that it is interesting to see how our codec fares against a highly optimized state-of-the-art method. For this comparison, we had to move to 16×16 rasterization (see Appendix B for details), and therefore we believe that these results are best presented in isolation. The result of the comparison can be seen in Table 2, which is an average for the three scenes rendered at 320×240. As can be seen, our 16×16 algorithm uses slightly less bandwidth and the quality is slightly better. On the face of it, it may seem as if the two 16×16 methods are more or less on par. However, the H.264 codec enjoys two advantages: First and most importantly, our 16×16 method is not allowed to predict across the 8×8 pixel boundaries within the 16×16 blocks (see Appendix B). This turned out very difficult to disable in the H.264 codec, and was thus allowed for the H.264 codec. Such prediction is highly valuable, and is generally regarded to be one of the major reasons why H.264 works so well not only as a video codec but also as a still-image codec. Second, the employed H.264 encoder uses rate-distortion optimization, effectively recoding the block multiple times and choosing the best trade-off between bit rate and quality. Such coding is computationally expensive, but performs well in terms of coding efficiency. Thus, given that our codec does not enjoy these two advantages, we find it notable to see that we are still getting slightly better results, and we think that our comparison shows that our technique is rather competitive against H.264 compression in the color buffer compression context. In the table, we have also included our 8×8 results, and as can be seen, this variant of our algorithm performs a further bit better. This is due to the fact that a system with 16×16 tiles will read and write several pixels that are never processed.

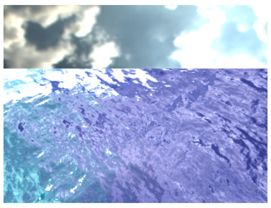
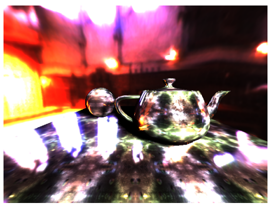
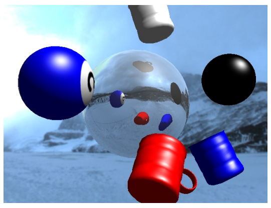
Scene	**Water**					**Shadows**					**Reflections**				
															
# triangles	44					6468					60336				
	A	**B**	**C**	**D**	**E**	**A**	**B**	**C**	**D**	**E**	**A**	**B**	**C**	**D**	**E**
Resolution	**320 × 240**														
Color BW	1.6	0.56	0.57	0.19	0.53	3.8	1.0	0.92	0.43	1.6	4.9	1.5	1.7	0.7	2.8
mPSNR (dB)	-	-	-	64.6	59.4	-	-	-	55.3	54.8	-	-	-	54.3	51.2
logERR[RGB]	-	-	-	0.026	0.029	-	-	-	0.029	0.032	-	-	-	0.028	0.038
HDR VDP	-	-	-	0.00	0.00	-	-	-	0.00	0.01	-	-	-	0.00	0.01
Compr factor	**1.0**	**2.9**	**2.9**	**8.7**	**3.1**	**1.0**	**3.8**	**4.1**	**8.8**	**2.3**	**1.0**	**3.3**	**2.9**	**6.8**	**1.7**
Resolution	**1024 × 768**														
Color BW	14.8	4.1	4.2	1.4	4.8	29.7	6.9	6.1	3.0	9.9	29.2	7.7	8.7	2.9	14.9
mPSNR (dB)	-	-	-	68.8	62.1	-	-	-	60.8	59.0	-	-	-	55.7	54.1
logERR[RGB]	-	-	-	0.021	0.023	-	-	-	0.015	0.019	-	-	-	0.024	0.027
HDR VDP	-	-	-	0.00	0.00	-	-	-	0.00	0.01	-	-	-	0.01	0.06
Compr factor	**1.0**	**3.6**	**3.5**	**10.5**	**3.1**	**1.0**	**4.3**	**4.8**	**9.8**	**3.0**	**1.0**	**3.8**	**3.4**	**10.1**	**2.0**

Table 1 Performance evaluation. The different columns are: **A** = uncompressed, **B** = Ström2008, **C** = our lossless, **D** = our lossy, **E** = OpenEXR B44A. The color buffer bandwidth (BW) is measured in MB/frame. Note that the parameters of our algorithm have been tuned so that the quality/error measures are approximately the same for column D and E. Since the quality is about the same, the compression factors can easily be compared. The HDR VDP rows show percentage of pixels where a human has a 75% chance of detecting an error. In general, one strives after a high mPSNR and low values on logRGB and HDR VDP.

6.6 Texture Compression using Buffer Compression

Most existing HDR texture compression schemes compress down to 8 bits per pixel, i.e., at a constant bit rate (CBR) per pixel. Out of curiosity, we experimented with the idea of using our lossy buffer compression algorithm for HDR texture compression at 8 bits per pixel as a final, smaller test. The performance in terms of quality was benchmarked against three state-of-art texture compression methods [12,15,20]. While our CBR-decompressor is exactly the same as before, the compressor needed some minor modifications to enable a CBR mode. Our approach is simple, and the core is a basic iterative search method to find one of 32 possible parameter sets, where a parameter set consists of seven parameters; flatness thresholds and quantization levels for each of the Y, C_o and C_g-components, and a restart quantization level. Parameter set 0 represents the mildest compression setting (highest quality, highest number of bits) with increasingly stronger compression settings as we move up towards parameter set 31, which represents the strongest compression setting (lowest quality, lowest number of bits). The chosen parameter set is signalled with five extra bits in the compressed tile data. The stopping criteria for the iterative search method is to test if the number of generated bits is within the range $512 - \tau_{\text{close}}$ and 512. 512 is the number of bits of an 8×8 pixel tile at 8 bits per pixel. τ_{close} is configurable and we used a value of 32 in our test. As an additional stopping criteria, there is a maximum iterations counter which was set to 6 in our experiment.

The search method works as follows: in the first iteration, the tile is encoded using the mid parameter set 15, which is the center of the range 0 to 31. If the first stop criteria is met, we stop and parameter set 15 is chosen. In the second iteration the next candidate parameter set is 23 (half way between 15 and 31) if the compressed tile size was above 512 bits, or 7 (halfway between 0 and 15) if it was below $512 - \tau_{\text{close}}$. Since we in the following iterations have at least two previous attempts, the remaining candidate parameter sets (3rd, 4th, 5th..) are calculated as the linear interpolation of the last two. The candidate parameter sets are clamped to be within the range $0-31$. If we cannot find a parameter set that reach the stop range $512 - \tau_{\text{close}}$, the maximum iteration counter halt the search. The chosen parameter set is then the best candidate (closest to, but below 512 bits) of the earlier attempts. For the test images in our experiment, the average number of iterations was around four. With this configuration, the execution time of our CBR-encoder was three orders of magnitude faster than that of Munkberg et al. [12]. Both implementations were written in non-optimzed C++. While our algorithm is almost symmetrical in terms of complexity, the algorithm proposed by Munkberg et al. represents a typical highly asymmerical algorithm.

The results can be seen in Table 3, and our algorithm performs quite well also as an HDR texture compressor. It should be noted, however, that this comparison is a bit unfair in that our algorithm operates on 8×8 pixel tiles, while the other algorithms operate on

Textures	mPSNR (dB)			
	Our	Sun 2008	Munkberg 2008	Roimela 2008
BigFogMap	50.8	51.0	51.9	50.4
Cathedral	37.5	39.7	40.0	34.3
Memorial	44.3	46.8	46.5	41.7
Room	46.9	48.1	48.6	44.0
Desk	39.7	41.5	40.3	28.4
Tubes	32.2	35.7	35.7	27.0

Table 3 Our algorithm used as a texture compressor/decompressor. Benchmarked against three state-of-the-art texture compression algorithms [12, 15, 20] for objective quality. Note that the values in this table for the three other codecs are copied from the paper by Sun et al. [20]. We have left out logRGB and HDR-VDP results since they indicate similar relations between the contenders. The images cann be found in Munkberg et al's paper [12].

smaller 4×4 tiles. For one thing, this triggers the high performance two-dimensional predictor for more pixels in the tile (49 out of 64 pixels (76.5%) for 8×8 compared to 9 out of 16 (56.2%) for the 4×4 methods). On the other hand, our algorithm was not designed as a texture compressor, but we thought it would be interesting to find out whether a single codec could work for both texture and buffer compression, and we believe that our experiment indicates that this is so.

7 Conclusions and Future Work

In this paper, we have presented the first lossy floating-point buffer compression algorithm, with results indicating virtually lossless image quality. The compression factors were between $2 - 3$ times larger than state-of-the-art lossless color buffer compression algorithms. If a bit more loss of image quality can be accepted, the compression gain can be much larger. Due to that computation capability continues to grow at a much faster pace than memory bandwidth and latency, we believe that our work can influence the overall performance substantially for future GPUs. We therefore hope our work will spur a renewed interest in high-dynamic range color buffer and texture compression.

As computations become less and less expensive in relation to memory bandwidth usage, it would be interesting to investigate computationally (very) expensive compression/decompression algorithms with better compression ratios and image quality. This is left for future work.

8 Supplementary Material: Animated content

In order to inspect that the lossy algorithm does not introduce additional artifacts during motion, an animated scene have been put together in a video. The video shows the uncompressed sequence of frames to the left and the lossy compression to the right. These frames have been rendered using the Shadow scene using the same compression parameter settings as the results in Table 1 but at 640×480 resolution. The color buffer bandwidth is thus about a factor of 9 lower than uncompressed mode (the left video) and about a factor of 2 lower than our algorithm in lossless mode.

Acknowledgements We acknowledge support from the Swedish Foundation for Strategic Research and Vinnova. Thanks to Rickard and Clinton from Ericsson Research for Python script hacking, and to Jon Hasselgren for coding help and reviewing. In addition, Tomas Akenine-Möller is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation.

References

1. A.C. Beers, M. Agrawala, and Navin Chadda. Rendering from Compressed Textures. In *Proceedings of ACM SIGGRAPH 96*, pages 373–378, 1996.
2. R. Bogart, F. Kainz, and D. Hess. OpenEXR Image File Format. In *ACM SIGGRAPH Sketches & Applications*, 2003.
3. Ahmet Caglar and Amin Ojani. Evaluation and Hardware Implementation of Real-Time Color Buffer Compression Algorithms. Master's thesis, Linköping University, 11 2008.
4. Solomon W. Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory*, (July):399–401, 1966.
5. Jon Hasselgren and Tomas Akenine-Möller. Efficient Depth Buffer Compression. In *Graphics Hardware*, pages 103–110, 2006.
6. Günter Knittel, Andreas G. Schilling, Anders Kugler, and Wolfgang Straßer. Hardware for Superior Texture Performance. *Computers & Graphics,*, 20(4):475–481, 1996.
7. Jaakko Lehtinen. A Framework for Precomputed and Captured Light Transport. *ACM Transactions on Graphics,*, 26(4):13, 2007.
8. P. Lindstrom and M. Isenburg. Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualisation and Computer Graphics,*, 12(5):1245–1250, 2006.
9. Henrique Malvar and Gary Sullivan. YCoCg-R: A Color Space with RGB Reversibility and Low Dynamic Range. In *JVT-I014r3*, 2003.
10. Rafał Mantiuk, Scott Daly, Karol Myszkowski, and Hans-Peter Seidel. Predicting Visible Differences in High Dynamic Range Images – Model and its Calibration. In *Human Vision and Electronic Imaging X*, pages 204–214, 2005.
11. Jacob Munkberg, Petrik Clarberg, Jon Hasselgren, and Tomas Akenine-Möller. High Dynamic Range Texture Compression for Graphics Hardware. *ACM Transactions on Graphics,*, 25(3):698–706, 2006.
12. Jacob Munkberg, Petrik Clarberg, Jon Hasselgren, and Tomas Akenine-Möller. Practical HDR Texture Compression. *Computer Graphics Forum*, 27(6):1664–1676, 2008.
13. Jim Rasmusson, Jon Hasselgren, and Tomas Akenine-Möller. Exact and Error-bounded Approximate Color Buffer Compression and Decompression. In *Graphics Hardware*, pages 41–48, 2007.
14. Robert F. Rice. Some Practical Universal Noiseless Coding Techniques. Technical Report 22, Jet Propulsion Lab, 1979.

15. K. Roimela, T. Aarnio, and J. Itäranta. Efficient High Dynamic Range Texture Compression. *Symposium on Interactive 3D Graphics and Games*, pages 207–214, 2008.
16. Kimmo Roimela, Tomi Aarnio, and Joonas Itäranta. High Dynamic Range Texture Compression. *ACM Transactions on Graphics,*, 25(3):707–712, 2006.
17. Jacob Ström and Tomas Akenine-Möller. iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones. In *Graphics Hardware*, pages 63–70, 2005.
18. Jacob Ström and Per Wennersten. Low-Bitrate Table-Based Alpha Compression. In *to appear in Eurographics*, 2009.
19. Jacob Ström, Per Wennersten, Jim Rasmusson, Jon Hasselgren, Jacob Munkberg, Petrik Clarberg, and Tomas Akenine-Möller. Floating-Point Buffer Compression in a Unified Codec Architecture. In *Graphics Hardware*, pages 96–101, 2008.
20. Wen Sun, Yan Lu, Feng Wu, and Shipeng Li. DHTC: an effective DXTC-based HDR texture compression scheme. In *Graphics Hardware*, pages 85–94, 2008.
21. J. Teuhola. Fast Image Compression by Quadtree Prediction. *Real-Time Imaging*, (4):299–308, 1998.
22. Jay Torborg and Jim Kajiya. Talisman: Commodity Realtime 3D Graphics for the PC. In *Proceedings of SIGGRAPH*, pages 353–364, 1996.
23. Lvdi Wang, Xi Wang, Peter-Pike Sloan, Li-Yi Wei, Xin Tong, and Baining Guo. Rendering from Compressed High Dynamic Range Textures on Programmable Graphics Hardware. In *Symposium on Interactive 3D Graphics and Games*, pages 17–24, 2007.
24. Roland Wilson. Quad-tree predictive coding: A new class of image data compression algorithms. *IEEE International Conference on Acoustics, Speech and Signal Processing*, 9:527–530, 1984.

9 Appendix

A OpenEXR B44A Lossy Compressor

The B44A encoder is implemented in OpenEXR, and is loosely based on the work by Roimela et al. [16]. In Section 6, we use this algorithm for our lossy compression comparisons, and hence include a more detailed description of the algorithm here. The B44A algorithm operates on the individual color channels of 4×4 blocks. The bit patterns are treated as integers, and the top left value is first stored in full 16 bits. Each pixel is then predicted from its left neighbor, except for the first pixel on each row, which is predicted from its upper neighbor. The prediction errors between the original and the predicted values are then formed. If all of these prediction errors are in the interval $[-32, 31]$, they are stored using six bits each. If not, the original values are divided by 2^k and rounded (starting with $k = 1$), and the prediction errors are formed again. If they still do not fit the $[-32, 31]$ interval, the process repeats, etc. For some k, the prediction errors will be small enough so that six bits is sufficient, and the block is then successfully compressed. After compression each channel contains the first value (16 bits), the k-value (6 bits) and the prediction errors (15 6-bit values), all in all 112 bits (14 bytes). However, using six bits to store the k-value allows for a division of 2^{63}, which will never be needed on a 16-bit number. Thus, the B44A encoder reserves this value for the case when the entire block is constant. In this case, only the first value (16 bits) and the shift value (6-bits) are needed, giving 22 bits or 3 bytes. Our version of the B44A encoder always assumes that the alpha component is 1.0, and hence only compresses 3 channels. Also, it compresses 8×8 blocks by concatenating the output from four 4×4 blocks. The resulting bit rate is therefore between $4 \cdot 3 \cdot 3 = 36$ bytes and $4 \cdot 14 \cdot 3 = 168$ bytes.

B H.264

H.264 is a state-of-the-art video codec that can also be used for compressing still images. While there exists no profile that can handle 15-bit color component depths, there is a 14-bit version. We have created a lossy coder based on H.264 by right-shifting our 15-bit data one step to 14 bits, basically destroying the least significant bit. The resulting 14-bit data is then compressed using the JM (v14.2) reference implementation of H.264 using the High 4:4:4 profile.

The smallest image size H.264 can handle is 16×16 pixels. It would have been possible to modify the H.264 codec so that it could compress 8×8 tiles (like the other algorithms in our comparison), but given the size and complexity of the H.264 code, we refrained from that option. Instead, we have chosen to change the rasterization block size from 8×8 to 16×16 for this comparison only, and hence the H.264 codec can be used unaltered. Our proposed algorithm is adapted to 16×16 pixels by calling our 8×8 compression function four times, which implies that no prediction is done between the four 8×8 blocks. The H.264 encoder, on the other hand, will make prediction across the 8×8 borders inside a 16×16 block, which gives the H.264 code an advantage.

We configured the H.264 codec to work in 4:4:4 mode, which means that no subsampling is used. The reason for this is that subsampling the chrominances gave highly disturbing artifacts along triangle edges. This is a consequence of the fact that H.264 does not have a special mode for prediction across edges.

Jim Rasmusson received his M.Sc. in Electrical Engineering from Lund University in 1989. Since 1990 he has been working in the mobile phone industry. Since 2007 he is part time pursuing a Ph.D. Thesis in 3D graphics in Tomas Akenine-Möller's group at Lund University. He also works for Ericsson Research. His main research interests are graphics, image and video processing for mobile phones.

Jacob Ström received his M.Sc. in Computer Science and Engineering from Lund University in 1995. In 1998 he received a Fulbright grant and spent one year at the MIT Media Lab in Boston as a visiting Ph.D. student. He received his Ph.D. in image coding from Linköping University in 2002, and is currently a Senior Specialist in Graphics and Image Processing at Ericsson Research in Stockholm, Sweden. Jacob has worked with mobile graphics, and together with Tomas Akenine-Möller he wrote the first SIGGRAPH publication on mobile graphics in 2003. Current interests include compression techniques for graphics, and he and Tomas received the best paper award at Graphics Hardware 2005 for the ETC texture compression scheme, which is now part of the OpenGL ES API.

Tomas Akenine-Möller received his M.Sc. in Computer Science and Engineering from Lund University in 1995, and a

Ph.D. in graphics from Chalmers University of Technology in 1998. He has worked on shadow generation, mobile graphics, wavelets, high-quality rendering, collision detection, and more. Tomas has several papers published at the ACM SIGGRAPH conference, and his first SIGGRAPH paper was on the pioneering topic of mobile graphics together with Jacob Ström in 2003. He co-authored the Real-Time Rendering book with Eric Haines and Naty Hoffman, and received the best paper award at Graphics Hardware 2005 with Jacob Ström for the ETC texture compression scheme, which is now part of the OpenGL ES API. Current research interests are in graphics hardware both for mobile devices and desktops, new computing architectures, collision detection, high-quality rapid rendering techniques, and many-core rendering.